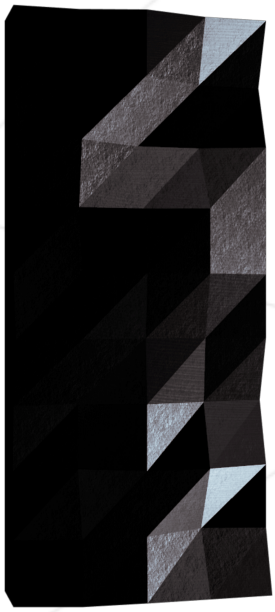


Code Nexus

Polymorphic Data Streams in the Digital Matrix

Summary: Enter the Code Nexus as a Stream Engineer! Master method overriding and subtype polymorphism while building advanced data processing pipelines that adapt and evolve in real-time through the digital matrix.

Version: 2.0

Contents

Chapter I

Foreword

Welcome to the Code Nexus, Stream Engineer!

The year is 2087. In the sprawling digital metropolis of Neo-Tokyo, data flows through quantum fiber networks like neon-bright rivers of pure information. The **Code Nexus** stands as humanity's greatest achievement—a cybernetic cathedral where billions of data streams converge, transform, and evolve in perfect harmony.

But the Nexus holds a secret that separates it from the crude data processors of the past: it doesn't just consume data—it **understands** it. Each data stream carries its own digital signature, its own behavioral patterns, its own electronic soul. Financial transactions pulse with the rhythm of global markets. Sensor readings whisper the secrets of environmental change. Neural network outputs sing with artificial consciousness.

How does a single system comprehend such diversity? Through the cybernetic principle of **polymorphism**—the art of creating digital chameleons that adapt their behavior while maintaining their core identity. In the old world, engineers built rigid, specialized systems. The Nexus transcends this limitation through **method overriding**, where the same processing node becomes a shapeshifter, handling any data stream while honoring its unique nature.

As a Stream Engineer in the Nexus, you'll master the forbidden knowledge of **inheritance hierarchies**—digital bloodlines that pass traits from parent to child while allowing each generation to evolve beyond its origins. You'll learn that overriding a method isn't just changing code—it's rewriting the genetic code of the digital organism itself.

The chrome towers of the Nexus await your expertise. Every data stream you engineer will be a testament to polymorphic design—where unified interfaces dance with specialized behaviors, creating systems that are both harmonious and infinitely adaptable.

Welcome to the future. Welcome to the Code Nexus.

Chapter II

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Introduction

NEXUS CLEARANCE LEVEL: STREAM ENGINEER INITIATE

The chrome spires of the Code Nexus pierce the neon-soaked sky of Neo-Tokyo, their quantum processors humming with the collective consciousness of a trillion data streams. You stand at the threshold of digital transcendence, ready to join the elite ranks of Stream Engineers who keep the Nexus alive.

Your Neural Interface Activation Sequence:

- **Phase Alpha**: `Data Processor Foundation` - Forge your first neural pathways with method overriding
- **Phase Beta**: `Polymorphic Streams` - Evolve adaptive data organisms through inheritance
- **Phase Gamma**: `Nexus Integration` - Architect the ultimate multi-stream consciousness

Each phase rewrites your digital DNA, teaching you to think not in rigid code, but in **living interfaces** and **evolving implementations**. By the final phase, you'll understand the Nexus's deepest secret: how a single consciousness can process infinite data forms while maintaining perfect digital harmony.

NEXUS CORE DIRECTIVE: This neural conditioning focuses on **method overriding** and **subtype polymorphism**. Your digital organisms must demonstrate how different classes share common neural pathways while expressing unique behavioral patterns through inheritance.

STREAM ENGINEER PROTOCOL: All Nexus operatives must achieve mastery of polymorphic design patterns. The digital matrix's survival depends on systems that evolve without fragmenting their core interfaces.

NEURAL DIAGNOSTIC SUITE: A main.py diagnostic program interfaces directly with your implementations. This neural probe will only achieve synchronization if you've properly architected all required digital organisms. Execute python3 main.py to verify your polymorphic constructs are achieving consciousness.

Chapter IV

Engineering Guidelines

IV.1 Nexus Standards

- Your project must be written in **Python 3.10 or later**.
- Your project must adhere to the **flake8** coding standard.
- **All code must include comprehensive type annotations** using the `typing` module.
- All classes must demonstrate proper **inheritance** relationships.
- Method overriding must be used purposefully to show **specialized behavior**.
- Exception handling should protect the data streams from corruption.
- Only standard library imports are authorized unless specified.
- Focus on demonstrating **polymorphic behavior** clearly in your implementations.

Required Type Annotations:
All code must include comprehensive type annotations using the `typing` module:

- Import required types: `from typing import Any, List, Dict, Union, Optional`
- Import ABC classes: `from abc import ABC, abstractmethod`
- All function parameters must have type annotations
- All function return types must be specified
- Class attributes should be typed where appropriate

Example: `def process(self, data:  Any) -> str:`

IV.2 Stream Engineering Principles

- **Interface Consistency**: Overridden methods must maintain the same signature as their parent methods.
- **Behavioral Specialization**: Each subclass should provide meaningful, distinct behavior.
- **Polymorphic Usage**: Demonstrate that different objects can be used interchangeably through common interfaces.
- **Inheritance Hierarchy**: Build logical class relationships that reflect real-world data processing concepts.

The Code Nexus operates on a simple principle: **same interface, different behavior.** When you call the same method on different objects, each should respond in its own specialized way while maintaining interface compatibility.

Chapter V

Exercise 0: Data Processor Foundation

	Exercise0
stream_processor	
Directory: *ex*0/	
Files to Submit: `stream_processor.py`	
Authorized: `print()`	

This exercise requires the use of classes with inheritance, the super() function, try/except blocks for error handling, and ABC (Abstract Base Class) with @abstractmethod decorators. Type hints from the typing module (Any, List, Dict, Union, Optional) must be used throughout.

Engineering Brief: Welcome to the Code Nexus! Build the foundation of our data processing system. You'll create the base processor architecture and demonstrate how different data types can share common processing interfaces while maintaining their unique characteristics.

Your Mission: Create a polymorphic data processing system that demonstrates method overriding. Build a base `DataProcessor` class and specialized processors for different data types.

System Architecture:

- **Base Class**: `DataProcessor` - an abstract base class defining the common processing interface

- **Specialized Classes**: `NumericProcessor()`, `TextProcessor()`, `LogProcessor()` (no constructor parameters required)
- **Required Methods** (must be implemented in all classes):
  - `process(self, data: Any) -> str` - Process the data and return result string
  - `validate(self, data: Any) -> bool` - Validate if data is appropriate for this processor
  - `format_output(self, result: str) -> str` - Format the output string
- **Polymorphic Behavior**: Same method calls, different specialized behaviors

Required Implementation:

- Create a `DataProcessor` abstract base class using `ABC` and `@abstractmethod`
- Mark `process()` and `validate()` as abstract methods
- Provide a default implementation for `format_output()` that can be overridden
- Override abstract methods in subclasses to provide specialized behavior
- Demonstrate polymorphic usage by processing different data types through the same interface
- Include proper error handling for invalid data

Focus on demonstrating how **method overriding** allows different processors to handle their specific data types while maintaining a consistent interface. This is the foundation of polymorphic design!

Example:

```
$> python3 stream_processor.py
=== CODE NEXUS - DATA PROCESSOR FOUNDATION ===

Initializing Numeric Processor...
Processing data: [1, 2, 3, 4, 5]
Validation: Numeric data verified
Output: Processed 5 numeric values, sum=15, avg=3.0

Initializing Text Processor...
Processing data: "Hello Nexus World"
Validation: Text data verified
Output: Processed text: 17 characters, 3 words

Initializing Log Processor...
Processing data: "ERROR: Connection timeout"
Validation: Log entry verified
Output: [ALERT] ERROR level detected: Connection timeout

=== Polymorphic Processing Demo ===
```

```
Processing multiple data types through same interface...
Result 1: Processed 3 numeric values, sum=6, avg=2.0
Result 2: Processed text: 12 characters, 2 words
Result 3: [INFO] INFO level detected: System ready

Foundation systems online. Nexus ready for advanced streams.
```

How does method overriding enable the same processing interface to handle completely different data types? What makes this approach more powerful than separate processing functions?

Chapter VI

Exercise 1: Polymorphic Streams

	Exercise1
data_stream	
Directory: *ex*1/	
Files to Submit: `data_stream.py`	
Authorized: `isinstance()`, `print()`	

This exercise requires the use of classes with inheritance, super(), try/except blocks, list comprehensions for data processing, and ABC with @abstractmethod. Type hints from the typing module must be used. The isinstance() function is needed for type checking.

Engineering Brief: Excellent foundation work! Your processor architecture. Now for the real challenge: building adaptive data streams that can handle multiple data types simultaneously while maintaining processing efficiency and type safety.

Your Mission: Create a sophisticated data streaming system that demonstrates advanced polymorphic behavior. Build stream handlers that can process mixed data types while maintaining type-specific optimizations.

Advanced Architecture:

- **Stream Base**: `DataStream` - an abstract base class with core streaming functionality
- **Specialized Streams**: `SensorStream(stream_id)`, `TransactionStream(stream_id)`, `EventStream(stream_id)`

- **Required Methods** (must be implemented in DataStream and overridden in subclasses):
  - `process_batch(self, data_batch: List[Any]) -> str` - Process a batch of data
  - `filter_data(self, data_batch: List[Any], criteria: Optional[str] = None) -> List[Any]` - Filter data based on criteria
  - `get_stats(self) -> Dict[str, Union[str, int, float]]` - Return stream statistics
- **Stream Manager**: `StreamProcessor` that handles multiple stream types polymorphically
- **Advanced Features**: Batch processing, filtering, transformation pipelines

Required Implementation:

- Create a `DataStream` abstract base class with `process_batch()` as an abstract method
- Provide default implementations for `filter_data()` and `get_stats()` that can be overridden
- Implement specialized stream classes with overridden behavior for different data domains
- Build a `StreamProcessor` that can handle any stream type through polymorphism
- Demonstrate batch processing of mixed stream types
- Include stream filtering and transformation capabilities
- Add comprehensive error handling for stream processing failures

This exercise demonstrates **subtype polymorphism** in action. Your StreamProcessor should be able to handle any DataStream subtype without knowing the specific implementation details. This is the power of polymorphic design!

Example:

```
$> python3 data_stream.py
=== CODE NEXUS - POLYMORPHIC STREAM SYSTEM ===

Initializing Sensor Stream...
Stream ID: SENSOR_001, Type: Environmental Data
Processing sensor batch: [temp:22.5, humidity:65, pressure:1013]
Sensor analysis: 3 readings processed, avg temp: 22.5°C

Initializing Transaction Stream...
Stream ID: TRANS_001, Type: Financial Data
Processing transaction batch: [buy:100, sell:150, buy:75]
Transaction analysis: 3 operations, net flow: +25 units

Initializing Event Stream...
Stream ID: EVENT_001, Type: System Events
Processing event batch: [login, error, logout]
Event analysis: 3 events, 1 error detected

=== Polymorphic Stream Processing ===
Processing mixed stream types through unified interface...

Batch 1 Results:
- Sensor data: 2 readings processed
- Transaction data: 4 operations processed
- Event data: 3 events processed

Stream filtering active: High-priority data only
Filtered results: 2 critical sensor alerts, 1 large transaction

All streams processed successfully. Nexus throughput optimal.
```

How does polymorphism allow the StreamProcessor to handle different stream types without knowing their specific implementations? What are the benefits of this design approach?

Chapter VII

Exercise 2: Nexus Integration

	Exercise2
nexus_pipeline	
Directory: *ex2/*	
Files to Submit: `nexus_pipeline.py`	
Authorized: `isinstance()`, `print()`, `collections`	

This exercise requires the use of classes with inheritance, super(), try/except blocks, list and dict comprehensions for data processing, ABC with @abstractmethod, and Protocol for duck typing. The collections module is authorized for advanced data structures. Type hints from typing module (including Protocol) must be used throughout.

Engineering Brief: Outstanding stream engineering! Your promotion to Senior Stream Engineer. Your final challenge: integrate everything into a complete data processing pipeline that demonstrates mastery of polymorphic architecture at enterprise scale.

Your Mission: Build the complete Code Nexus data processing pipeline—a sophisticated system that combines multiple processing stages, handles complex data transformations, and demonstrates advanced polymorphic patterns used in real-world data engineering.

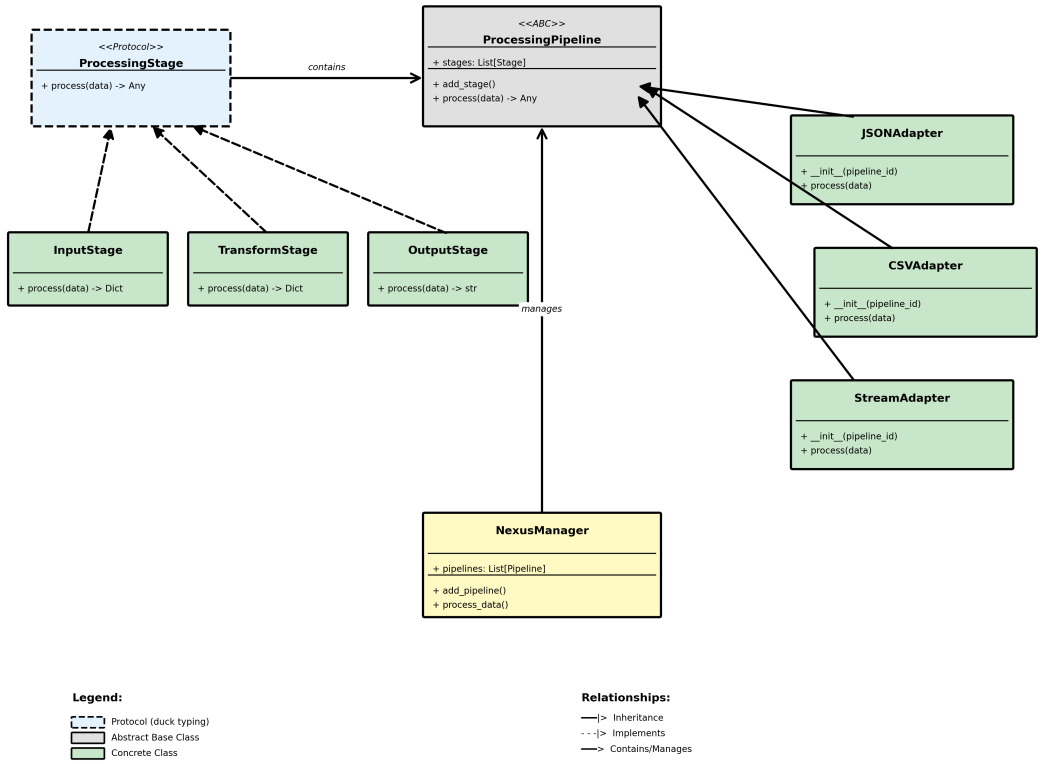
Enterprise Architecture:

- **Pipeline Base**: `ProcessingPipeline` - an abstract base class with configurable stages

- **Processing Stages**: `InputStage()`, `TransformStage()`, `OutputStage()` (no constructor parameters required)
  - Each stage must implement: `process(self, data: Any) -> Any`
- **Data Adapters**: `JSONAdapter(pipeline_id)`, `CSVAdapter(pipeline_id)`, `StreamAdapter(pipeline_id)`
  - Each adapter must override: `process(self, data: Any) -> Union[str, Any]`
- **Pipeline Manager**: `NexusManager` orchestrating multiple pipelines
- **Advanced Features**: Pipeline chaining, error recovery, performance monitoring

Class Diagram - Architecture Overview:

The following UML class diagram illustrates the complete architecture, showing inheritance relationships, composition patterns, and the distinction between Protocol (duck typing) and ABC (abstract base class). This diagram clarifies which classes inherit from which, and how stages relate to pipelines.



Architecture Explanation:

- **ProcessingStage (Protocol)**: Interface for stages using duck typing. Any class with `process()` can act as a stage.
- **ProcessingPipeline (ABC)**: Abstract base managing stages. Contains a list of stages and orchestrates data flow.
- **Stage Classes**: `InputStage`, `TransformStage`, `OutputStage` implement the Protocol (duck typing, no inheritance). No constructor parameters.
- **Adapter Classes**: `JSONAdapter`, `CSVAdapter`, `StreamAdapter` inherit from `ProcessingPipeline` and override `process()`. Each takes `pipeline_id` parameter.
- **NexusManager**: Orchestrates multiple pipelines polymorphically.
- **Key Relationships**: Composition (contains), Inheritance (inherits from), Implementation (implements protocol).

Required Implementation:

- Create a flexible `ProcessingPipeline` abstract base class with configurable processing stages
- Use `Protocol` or duck typing for stage interfaces (stages must have a `process()` method)
- Implement specialized pipeline stages (InputStage, TransformStage, OutputStage) with `process()` methods
- Build data adapters (JSONAdapter, CSVAdapter, StreamAdapter) that inherit from ProcessingPipeline
- Each adapter should override the `process()` method for format-specific handling
- Create a `NexusManager` that orchestrates multiple pipelines polymorphically
- Demonstrate pipeline chaining where output from one pipeline feeds into another
- Include comprehensive error handling and recovery mechanisms
- Add performance monitoring and pipeline statistics

This is your masterpiece! Demonstrate how **method overriding** and **subtype polymorphism** enable building complex, maintainable systems. Your pipeline should handle any data type or processing requirement through polymorphic interfaces.

Example:

```
$> python3 nexus_pipeline.py
=== CODE NEXUS - ENTERPRISE PIPELINE SYSTEM ===

Initializing Nexus Manager...
Pipeline capacity: 1000 streams/second

Creating Data Processing Pipeline...
Stage 1: Input validation and parsing
Stage 2: Data transformation and enrichment
Stage 3: Output formatting and delivery

=== Multi-Format Data Processing ===

Processing JSON data through pipeline...
Input: {"sensor": "temp", "value": 23.5, "unit": "C"}
Transform: Enriched with metadata and validation
Output: Processed temperature reading: 23.5°C (Normal range)

Processing CSV data through same pipeline...
Input: "user,action,timestamp"
Transform: Parsed and structured data
Output: User activity logged: 1 actions processed

Processing Stream data through same pipeline...
Input: Real-time sensor stream
Transform: Aggregated and filtered
Output: Stream summary: 5 readings, avg: 22.1°C

=== Pipeline Chaining Demo ===
Pipeline A -> Pipeline B -> Pipeline C
Data flow: Raw -> Processed -> Analyzed -> Stored

Chain result: 100 records processed through 3-stage pipeline
Performance: 95% efficiency, 0.2s total processing time

=== Error Recovery Test ===
Simulating pipeline failure...
Error detected in Stage 2: Invalid data format
Recovery initiated: Switching to backup processor
Recovery successful: Pipeline restored, processing resumed

Nexus Integration complete. All systems operational.
```

How does the combination of method overriding and subtype polymorphism enable building scalable, maintainable data processing systems? What real-world engineering problems does this approach solve?

Chapter VIII

Turn in and Submission

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.

During evaluation, you may be asked to explain polymorphic behavior, demonstrate method overriding, extend your systems with new data types, or modify processing behavior. Make sure you understand how inheritance enables code reuse while allowing behavioral specialization.

You need to return only the files requested by the subject of this project. Focus on clean, well-documented code that clearly demonstrates mastery of method overriding and subtype polymorphism principles.